The following paper was originally published in the
Proceedings of the USENIX 1997 Annual Technical Conference
Anaheim, California, January 6-10, 1997

Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files

Gregory R. Ganger and M. Frans Kaashoek
Laboratory for Computer Science
Massachusetts Institute of Technology

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files

Gregory R. Ganger and M. Frans Kaashoek
M.I.T. Laboratory for Computer Science
Cambridge MA 02139, USA
{ganger,kaashoek}@lcs.mit.edu
http://www.pdos.lcs.mit.edu/

Abstract

Small file performance in most file systems is limited by slowly improving disk access times, even though current file systems improve on-disk locality by allocating related data objects in the same general region. The key insight for why current file systems perform poorly is that locality is insufficient — exploiting disk bandwidth for small data objects requires that they be placed adjacently. *We describe C-FFS (Co-locating Fast File System), which introduces two techniques,* **embedded inodes** *and* **explicit grouping**, *for exploiting what disks do well (bulk data movement) to avoid what they do poorly (reposition to new locations). With embedded inodes, the inodes for most files are stored in the directory with the corresponding name, removing a physical level of indirection without sacrificing the logical level of indirection. With explicit grouping, the data blocks of multiple small files named by a given directory are allocated adjacently and moved to and from the disk as a unit in most cases. Measurements of our C-FFS implementation show that embedded inodes and explicit grouping have the potential to increase small file throughput (for both reads and writes) by a factor of 5–7 compared to the same file system without these techniques. The improvement comes directly from reducing the number of disk accesses required by an order of magnitude. Preliminary experience with software-development applications shows performance improvements ranging from 10–300 percent.*

1 Introduction

It is frequently reported that disk access times have not kept pace with performance improvements in other system components. However, while the time required to fetch the first byte of data is high (i.e., measured in milliseconds), the subsequent data bandwidth is reasonable ($>$ 10 MB/second). Unfortunately, although file systems have been very successful at exploiting this bandwidth for large files [Peacock88, McVoy91, Sweeney96], they have failed to do so for small file activity (and the corresponding metadata activity). Because most files are small (e.g., we observe that 79% of all files on our file servers are less than 8 KB in size) and most files accessed are small (e.g., [Baker91] reports that 80% of file accesses are to files of less than 10KB), file system performance is often limited by disk access times rather than disk bandwidth.

One approach often used in file systems like the fast file system (FFS) [McKusick84] is to place related data objects (e.g., an inode and the data blocks it points to) near each other on disk (e.g., in the same cylinder group) in order to reduce disk access times. This approach can successfully reduce the seek time to just a fraction of that for a random access pattern. Unfortunately, it has some fundamental limitations. First, it affects only the seek time component of the access time[1], which generally comprises only about half of the access time even for random access patterns. Rotational latency, command processing, and data movement, which are not reduced by simply placing related data blocks in the same general area, comprise the other half. Second, seek times do not drop linearly with seek distance for small distances. Seeking a single cylinder (or just switching between tracks) generally costs a full millisecond, and this cost rises quickly for slightly longer seek distances [Worthington95]. Third, it is successful only when no other activity moves the disk arm between related requests. As a result, this approach is generally limited to providing less than a factor of two improvement in performance (and often much less).

Another approach, the log-structured file system

[1] We use the terms *access time* and *service time* interchangeably to refer to the time from when the device driver initiates a read or write request to when the request completion interrupt occurs.

(LFS), exploits disk bandwidth for all file system data, including large files, small files, and metadata. The idea is to delay, remap and cluster all modified blocks, only writing large chunks to the disk [Rosenblum92]. Assuming that free extents of disk blocks are always available, LFS works extremely well for write activity. However, the design is based on the assumption that file caches will absorb all read activity and does not help in improving read performance. Unfortunately, anecdotal evidence, measurements of real systems (e.g., [Baker91]), and simulation studies (e.g., [Dahlin94]) all indicate that main memory caches have not eliminated read traffic.

This paper describes the co-locating fast file system (C-FFS), which introduces two techniques for exploiting disk bandwidth for small files and metadata: *embedded inodes* and *explicit grouping*. Embedding inodes in the directory that names them (unless multiple directories do so), rather than storing them in separate inode blocks, removes a physical on-disk level of indirection without sacrificing the logical level of indirection. This technique offers many advantages: it halves the number of blocks that must be accessed to open a file; it allows the inodes for all names in a directory to be accessed without requesting additional blocks; it eliminates one of the ordering constraints required for integrity during file creation and deletion; it eliminates the need for static (over-)allocation of inodes, increasing the usable disk capacity [Forin94]; and it simplifies the implementation and increases the efficiency of explicit grouping (there is a synergy between these two techniques).

Explicit grouping places the data blocks of multiple files at adjacent disk locations and accesses them as a single unit most of the time. To decide which small files to co-locate, C-FFS exploits the inter-file relationships indicated by the name space. Specifically, C-FFS groups files whose inodes are embedded in the same directory. The characteristics of disk drives have reached the point that accessing several blocks rather than just one involves a fairly small additional cost. For example, even assuming minimal seek distances, accessing 16 KB requires only 10% longer than accessing 8 KB, and accessing 64 KB requires less than twice as long as accessing a single 512-byte sector. Further, the relative cost of accessing more data has been dropping over the past several years and should continue to do so. As a result, explicit grouping has the potential to improve small file performance by an order of magnitude over conventional file system implementations. Because the incremental cost is so low, grouping will improve performance even when only a fraction of the blocks in a group are needed.

Figure 1 illustrates the state-of-the-art and the improvements made by our techniques. Figure 1A shows the ideal layout of data and metadata for five single-block files, which might be obtained if one uses a fresh FFS partition. In this case, the inodes for all of the files are located in the same inode block and the directory block and the five file blocks are stored adjacently. With this layout, the prefetching performed by most disks will exploit the disk's bandwidth for reads and scatter/gather I/O from the file cache can do so for writes. Unfortunately, a more realistic layout of these files for an FFS file system that has been in use for a while is more like that shown in Figure 1B. Reading or writing the same set of files will now require several disk accesses, most of which will require repositioning (albeit with limited seek distances, since the picture shows only part of a single cylinder group). With embedded inodes, one gets the layout shown in Figure 1C, wherein the indirection between on-disk directory entries and on-disk inodes is eliminated. Finally, with both embedded inodes and ex-

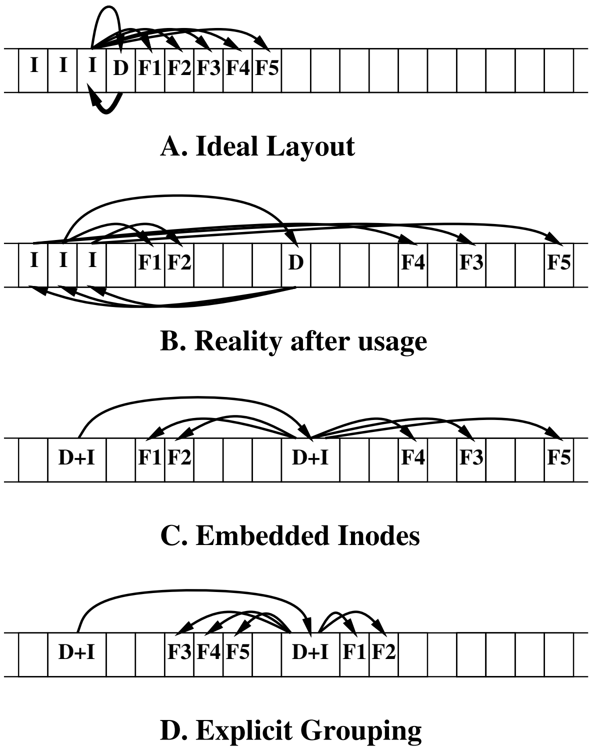


Figure 1: Organization and layout of file data on disk. This figure shows the on-disk locations of directory blocks (marked 'D'), inode blocks ('I') and the data for five single-block files ('F1' – 'F5') in four different scenarios: (A) the ideal conventional layout, (B) a more realistic conventional layout, (C) with the addition of embedded inodes, and (D) with both embedded inodes and explicit grouping (with a maximum group size of four blocks).

plicit grouping, one gets the layout shown in Figure 1D. In this case, one can read or write all five files with two disk accesses. Further, for files F1 and F2, one can read or write the name, inode and data block all in a single disk request. In our actual implementation, the maximum group size is larger than that shown and the allocation code would try to place the second group immediately after the first.

We have constructed a C-FFS implementation that includes both of these techniques. Measurements of C-FFS as compared to the same file system without these techniques show that, for small file activity, embedded inodes and explicit grouping can reduce the number of disk accesses by more than an order of magnitude. On the system under test (a modern PC), this translates into a performance increase of a factor of 5–7 in small file throughput for both reads and writes. Although our evaluation is preliminary, experiments with actual applications show performance improvements ranging from 10–300 percent.

The remainder of this paper is organized as follows: Section 2 provides more detailed motivation for co-location, by exploring the characteristics of modern disk drives and file storage usage. Section 3 describes our implementation of embedded inodes and explicit grouping. Section 4 shows that co-location improves performance significantly by comparing two file system implementations that differ only in this respect. Section 5 discusses related work. Section 6 discusses some open questions. Section 7 summarizes this paper.

2 Motivation

The motivating insights for this work fall into two broad categories: (1) The performance characteristics of modern disk drives, the de facto standard for on-line storage, force us to aggressively pursue adjacency of small objects rather than just locality. (2) The usage and organizational characteristics of popular file systems both suggest the logical relationships that can be exploited and expose the failure of current approaches in placing related data adjacently.

2.1 Modern Disk Drive Characteristics

It has repeatedly been pointed out that disk drive access times continue to fall behind relative to other system components. However, disk drive manufacturers have not been idle. They have matched the improvement rates of other system components in areas other than the access time, such as reliability, cost per byte, and recording density. Also, although it has not improved quite as rapidly, bulk data bandwidth has improved significantly. This section uses characteristics of modern disk drives to show that reading or writing several 4KB or 8KB disk blocks costs a relatively small amount more than reading or writing only one (e.g., 10% for 8KB extra and 100% for 56KB extra), and that this incremental cost is dropping. Because of the high cost of accessing a single block, it makes sense to access several even if only some of them are likely to be necessary. Readers who do not need to be convinced are welcome to skip to Section 2.2.

The service time for a disk request can be broken into two parts, one that is dependent on the amount of data being transferred and one that is not. The former usually consists of just the media transfer time (unless the media transfer occurs separately, as with prefetch or write-behind). Most disks overlap the bus transfer time with the positioning and media transfer times, such that what remains, the ramp-up and ramp-down periods, is independent of the size. Most of the service time components, most notably including command processing overheads, seek times, and rotational latencies, are independent of the request size. With modern disk drives and small requests, the size-independent parts of the service time dominate the size-dependent part (e.g., $>$ 90% of the total for random 8KB requests).

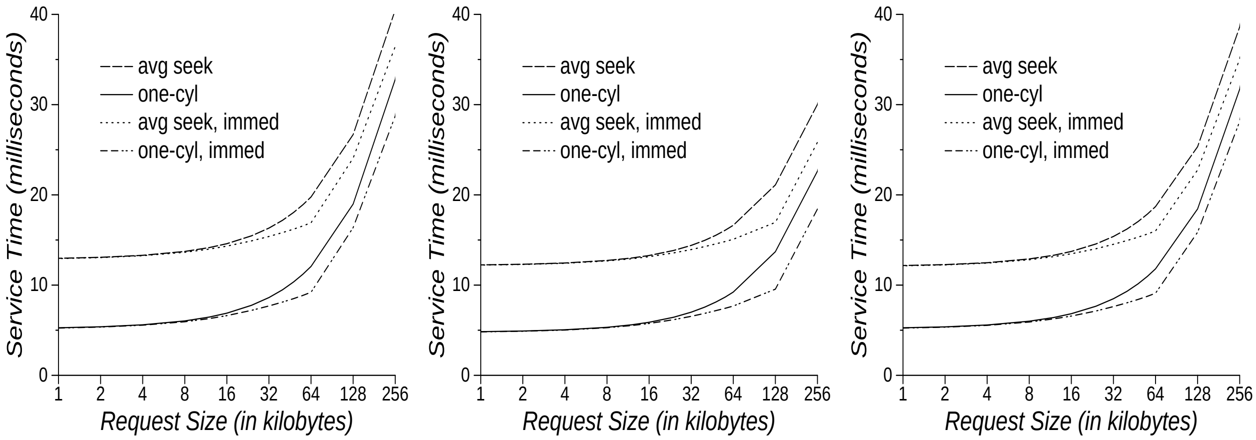
Another way to view the same two aspects of the service time are as per-request and per-byte. The dominating performance characteristic of modern disk drives is that the per-request cost is much larger than the per-byte cost. Therefore, transferring larger quantities of useful data in fewer requests will result in a significant performance increase.

One approach currently used by many file system implementors is to try to place related data items close to each other on the disk, thereby reducing the per-request cost. This approach does improve performance, but not nearly as much as one might hope. It is generally limited to reducing seek distances, and thereby seek times, which represent only about half of the per-request time. Rotational latency, command processing and data movement comprise the other half. In addition, even track switches and single-cylinder seeks require a significant amount of time (e.g., a millisecond or more) because they still involve mechanical movement and settling delays. Further, this approach is successful only when no other activity uses the disk (and thereby moves the disk arm) between related requests. As a result, this approach is generally limited to providing less than a factor of two in performance (and in practice much less).

To help illustrate the above claims, Table 1 lists characteristics for three state-of-the-art (for 1996) disk drives [HP96, Quantum96, Seagate96]. Figure 2 shows, for the same three drives, average access times as a function of the request size. Several points can be inferred from these graphs. First, the incremental cost of reading or

Disk Drive Specification	Hewlett-Packard C3653a	Seagate Barracuda	Quantum Atlas II
Capacity	8.7 GB	9.1 GB	9.1 GB
Cylinders	5371	5333	5964
Surfaces	20	20	20
Sectors per Track	124–173	153–239	108–180
Rotation Speed	7200 RPM	7200 RPM	7200 RPM
Head Switch	< 1 ms	N/A	N/A
One-cyl. Seek	< 1 ms	0.6 ms (0.5 ms)	1.0 ms
Average Seek	8.7 ms (0.8 ms)	8.0 ms (1.5 ms)	7.9 ms
Maximum Seek	16.5 ms (1.0 ms)	19.0 ms (1.0 ms)	18.0 ms

Table 1: Characteristics of three modern disk drives, taken from [HP96, Seagate96, Quantum96]. N/A indicates that the information was not available. For the seek times, the additional time needed for write settling is shown in parentheses, if it was available.



(a) HP C3653a (b) Seagate Barracuda ST19171FC (c) Quantum Atlas II

Figure 2: Average service time as a function of request size. The X-axis uses log scale. Two of the lines in each plot represent the access time assuming an average seek time and a single-cylinder seek time, respectively. The other two lines in each plot (labelled "immed") represent the same values, but assuming that the disk utilizes a technique known as immediate or zero-latency access. The basic idea is to read or write the data sectors in the order that they pass under the read/write head rather than in strictly ascending order. This can eliminate part or all of the rotational latency aspect of the service time. These values were calculated from the data in Table 1 using the read seek times (which are shorter), assuming zero command initiation/processing overheads, and using the sector/track value for the outermost zone.

writing several blocks rather than just one is small. For example, a 16 KB access takes less than 10% longer than an 8 KB access, even assuming a minimal seek time. A 64 KB access takes less than twice as long as an 8 KB access. Second, immediate or zero-latency access extends the range of request sizes that can be accessed for small incremental cost, which will increase the effectiveness of techniques that co-locate multiple small data objects. Third, the seek time for random requests represents a little more than half the total service time. Therefore, eliminating it completely for a series of requests can halve each service time after the first. In comparison, co-location will eliminate the entire service time for a set of requests after paying only a slightly higher cost for the first. So, while reducing seek times can improve performance somewhat, aggressive co-location has the potential to provide much higher returns.

Not only are per-byte costs small relative to per-request costs, but they have been decreasing and are likely to continue to do so. For example, the HP C2247 disk drive [HP91, HP92] of a few years ago had only half as many sectors on each track as the HP C3653 listed in Table 1, but an average access time that was only 33% higher. As a result, a request of 64KB, which takes only 2 times as long as a request of 8KB on state-of-the-art disks, took nearly 3 times as long as a request of 8KB on older drives. Projecting this trend into the future suggests that co-location will become increasingly important.

2.2 File System Characteristics

File systems use metadata to organize raw storage capacity into human-readable name spaces. Most employ a hierarchical name space, using directory files to translate components of a full file name to an identifier for either the desired file or the directory with which to translate the next name component. At each step, the file system uses the identifier (e.g., an inode number in UNIX file systems) to determine the location of the file's metadata (e.g., an inode). This metadata generally includes a variety of information about the file, such as the last modification time, the length, and pointers to where the actual data blocks are stored. This organization involves several levels of indirection between a file name and the corresponding data, each of which can result in a separate disk access. Additionally, the levels of indirection generally cause each file's data to be viewed as an independent object (i.e., logical relationships between it and other files are only loosely considered).

To get a better understanding of real file storage organizations, we constructed a small utility to scan some of our group's file servers. The two SunOS 4.1 servers scanned supply 13.8 GB of storage from 9 file systems on 5 disks. At the time of the examination, 9.8 GB (71% of the total available) was allocated to 466,271 files.

In examining the statistics collected, three observations led us to explore embedded inodes and explicit grouping. First, as shown in Figure 3 as well as by previous studies, most files are small (i.e., 79% of those on our server are smaller than a single 8KB block). In addition to this static view of file size distributions, studies of dynamic file system activity have reported similar behavior. For example, [Baker91] states that, although most of the bytes accessed are in large files, "the vast majority of file accesses are to small files" (e.g., about 80% of the files accessed during their study were less than 10KB). These data, combined with the fact that

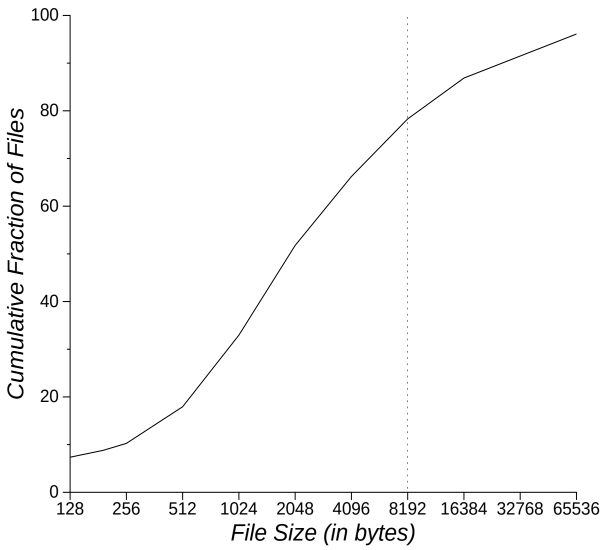
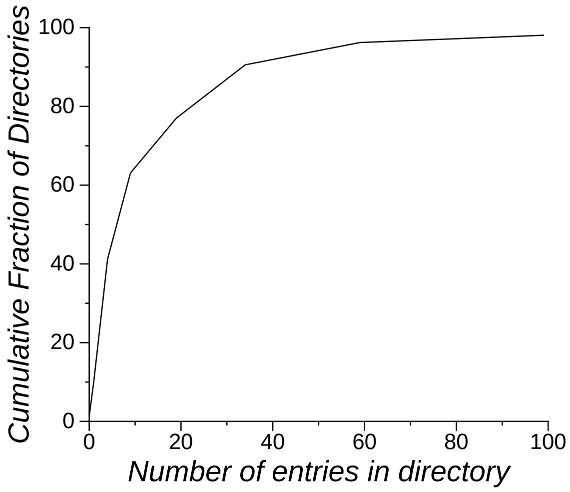


Figure 3: Distribution of file sizes measured on our file servers.

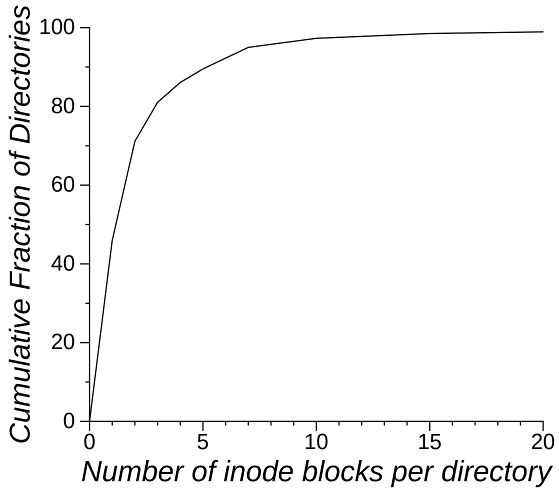
file system implementors are very good at dealing with large files, suggest that it is important to address the performance of small file access.

Second, despite the fact that each inode block contains 64 inodes, multiple inode blocks are often used to hold the metadata for files named in a particular directory (see Figure 4). On average, we found that the ratio of names in a directory to referenced inode blocks is roughly six to one. That is, every sixth name translates to a different inode block. This suggests that the current mechanism for choosing where to place an inode (i.e., inode allocation) does a poor job of placing related inodes adjacently. With embedded inodes, we store inodes in directories rather than pointers to inodes, except in the rare case (less than 0.1 percent, on our server) of having multiple links to a file from separate directories. In addition to eliminating a physical level of indirection, embedding inodes reduces the number of blocks that contain metadata for the files named by a particular directory.

Third, despite the fact that the allocation algorithm for single-block files in a directory might be expected to achieve an ideal layout (as in Figure 1A) on a brand new file system, these blocks tend to be local (e.g., in the same cylinder group) but not adjacent after the file system has been in use. For example, looking at entries in their physical order in the directory, we find that only 30% of directories are adjacent to the first file that they name. Further, fewer than 45% of files are placed adja-



(a) Directory Entries



(b) Inode Blocks

Figure 4: Distributions of the number of entries per directory and the corresponding number of inode blocks referred to on our file servers.

cent to the file whose name precedes their name in the directory.[2] The consequence of this lack of adjacency is that accessing multiple files in a directory involves disk head movement for every other file (when the file cache does not capture the accesses). The lack of adjacency occurs because, for the first block of a file, the file system always begins looking for free space at the beginning of the cylinder group. With explicit grouping, new file blocks for small files will be placed adjacently to other files in the same directory whenever possible. This should represent a substantial reduction in the number of disk requests required for a set of files named by a given directory.

We believe that the relationships inherent in the name space can be exploited to successfully realize the performance potential of modern disks' bandwidths. For example, many applications examine the attributes (i.e., reference the inodes) of files named by a given directory soon after scanning the directory itself (e.g., revision control systems that use modification times to identify files in the source tree that might have changed and directory listing programs that format the output based on permissions). Such temporal locality can be exploited directly with embedded inodes. Similarly, it is common for applications to access multiple files in a directory (e.g., when scanning a set of files for a particular string or when compiling and linking a multi-file program), as opposed to accessing files randomly strewn throughout the name space. Therefore, explicitly grouping small files in a directory represents a significant performance advantage.

3 Design and Implementation

To better exploit the data bandwidth provided by modern disk drives, C-FFS uses embedded inodes and explicit grouping to co-locate related small data objects. This section describes the various issues that arise when applying these techniques and how our C-FFS implementation addresses them.

3.1 Embedded Inodes

Conceptually, embedding inodes in directories is straightforward. One simply eliminates the previous inode storage mechanism and replaces the inode pointer generally found in each directory entry with the inode itself. Unfortunately, a few complications do arise: (1) Finding the location of an arbitrary inode given an

[2]Different applications access files named by a directory in different orders. For example, many shell utilities re-order file listings alphabetically, while multi-file compile and link programs often apply a user-specified ordering. As a result, we also looked at file adjacency under alternative file sequences. For example, in alphabetical order, only 16% of directories are adjacent to their first file and only 30% of files are adjacent to their predecessor. Even with the best-case ordering (by ascending disk block address), only 64% of files are next to their predecessor.

inode number (and to avoid changing other file system components, this file ID must remain unique) must still be both possible and efficient. (2) As part of enabling the discovery of an arbitrary inode's location, the current C-FFS implementation replaces inode numbers with inode *locators* that include an ID for the containing directory. One consequence of this approach is that when a file is moved from one directory to another, its inode locator will change. This can cause difficulties for system components that are integrated with the file system and rely on inode number constancy. (3) Although files with links from multiple directories are rare, they are useful and should still be supported. Also, files with multiple links from within a directory should be supported with low cost, since some applications (such as editors) use such links for backup management. (4) File system recovery after both system crashes and partial media corruption should be no less feasible than with a conventional file system organization.

One of our implementation goals (on which we were almost successful) was to change nothing beyond the directory entry manipulation and on-disk inode access portions of the file system. This section describes how C-FFS implements embedded inodes, including how C-FFS addresses each of the above issues.

Finding specific inodes

By eliminating the statically allocated, directly indexed set of inodes, C-FFS breaks the direct translation between inode number and on-disk inode location. To regain this, C-FFS replaces conventional inode numbers with inode *locators* and adds an additional on-disk data structure, called the *directory map*. In our current implementation, an inode locator has three components: a directory number, a sector number within the directory and an identifier within the sector. C-FFS currently sizes these at 16 bits, 13 bits and 3 bits, respectively, providing more than an order of magnitude more room for growth in both directory count and directory size than appears to be necessary from the characteristics of our file server. However, shifting to a 64-bit inode locator would overcome any realistic limitations.

To find the inode for an arbitrary inode locator, C-FFS uses the following procedure:

1. Look in the inode cache.

2. If the inode is not in the cache, use the directory number portion of the inode locator to index into the directory map. This provides the inode locator for the directory. A special directory number, 0, is used to refer to the "directory" containing multi-linked inodes that are not embedded (see below).

3. Get the directory's inode. This may require recursively executing steps 1, 2, and 3 several times. However, the recursion is always bounded by the root of the directory hierarchy and should rarely occur in practice, since getting to the file's inode locator in the first place required looking in the directory. Further, this recursion could be eliminated by exploiting the auxiliary physical location information included in the directory map (see below).

4. Read the directory block that contains the sector referred to by the sector number component of the inode number. (Note that the sector number is not really necessary — it simply allows us to restrict the linear scan.)

5. Traverse the set of directory entries in the sector to find the matching inode information, if it exists.

Moving files

Because C-FFS stores inodes inside the directories that name them, moving a file from one directory to another involves also moving the inode. Fortunately, this is easy. However, because inode locators encode the inode's location, including the containing directory's number, the inode locator must be changed when an inode moves.[3] In addition to file movement, C-FFS moves inodes when they are named by multiple directories (see below) and when the last name for a file is removed while it is still open (to support POSIX semantics). Unfortunately, changing the externally visible file ID can cause problems for system components, such as an NFS server, that relies on constancy of the file ID for a given file. Some solutions to this problem that we have considered include not actually moving the inode (and using the external link entry type discussed below), keeping an additional structure to correlate the old and new inode locators and forcing other system components to deal with the change. We currently use this last solution, but are growing concerned that too many applications expect file IDs to remain constant.

We currently believe that the correct solution is to bring back constant inode numbers and to introduce another table to translate inode numbers to inode locators. Properly implemented, we believe that such a table could be maintained with low cost. In particular, it would only need to be read when a desired inode is not in the inode cache. Also, by keeping both the inode number and the

[3] Note that moving a directory will change the inode locator for the directory but will not change the inode locators for any of the files it names. The inode locators for these latter files encode the directory's identity by its index into the directory map rather than by its inode locator.

inode locator in each inode, the table could be considered soft state that is reconstructed if the system crashes.

Supporting hard links

Although it is rare to have multiple hard links to a single file, it can be useful and we want to continue to allow it. Therefore, C-FFS replaces the inode pointer field of conventional directories with a type that can take on one of five values: (1) *invalid*, which indicates that the directory entry is just a space holder; (2) *embedded*, which indicates that the inode itself is in the entry; (3) *directory pointer*, which indicates that the entry contains a directory number that can be used to index into the directory map to find the corresponding inode locator (this entry type is used only for the special '.' and '..' entries); (4) *internal link*, which indicates that the entry contains a pointer to the location of the actual inode elsewhere within the directory; and (5) *external link*, which says that the entry contains a pointer to the location of the inode outside of the directory. Additional space in the directory entry is used to hold the inode or pointer for the latter four cases.

As suggested by the existence of an external link type, C-FFS stores some inodes outside of directories. In particular, C-FFS does this for inodes that are named by multiple files and for inodes that are not named but have been opened by some process. For the latter case, externalizing an inode is simple because we don't need to guarantee its existence across a system failure. Externalizing an inode to add a second external link, on the other hand, is expensive in our current C-FFS implementation, requiring two synchronous writes (one to copy the inode to its new home and one to update the directory from which is was moved). Externalized inodes are kept in a dynamically-growable, file-like structure that is similar to the IFILE in BSD-LFS [Seltzer93]. Some differences are that the externalized inode structure grows as needed but does not shrink and its blocks do not move once they have been allocated.

File system recovery

One concern that re-arranging file system metadata raises is that of integrity in the face of system failures and media corruption. Regarding the first, we have had no difficulties constructing an off-line file system recovery program much like the UNIX FSCK utility [McKusick94]. Although inodes are no longer at statically determined locations, they can all be found (assuming no media corruption) by following the directory hierarchy. We also do not believe that embedded inodes increase the time required to complete failure recovery, since reading the directories is required for checking link counts anyway. In fact, embedded inodes reduce the effort involved with verifying the link counts, since extra state need not be kept (a valid embedded inode has a link count of one plus the number of internal links).

The one potential problem that embedding inodes introduces is that an unfortunate media corruption can cause all files below a corrupted directory to be lost. Even though they are uncorrupted, the single way to find them will have been destroyed. This is unacceptable when compared to the current scheme, which loses a maximum of 64 files or directories when an inode block is destroyed. All files that become disconnected from the name hierarchy due to such a loss can still be found. Fortunately, we can employ a simple solution to this problem — redundancy. By augmenting the directory map with the physical location of each directory's inode, we eliminate the loss of directories and files below a lost directory. Although the absolute amount of loss may be slightly higher (because the density of useful information is higher), it should be acceptable given the infrequency of post-factory media corruption.

Simplifying integrity maintenance

Although the original goal of embedded inodes was to reduce the number of separate disk requests, a pleasant side effect is that we can also eliminate one of the sequencing constraints associated with metadata updates [Ganger94]. In particular, by eliminating the physical separation between a name and the corresponding inode, C-FFS exploits a disk drive characteristic to atomically update the pair. Most disks employ powerful error correcting codes on each sector, which has the effect of eliminating (with absurdly high probability) the possibility of only part of the sector being updated. So, by keeping the two items in the same sector, we can guarantee that they will be consistent with respect to each other. For file systems that use synchronous writes to ensure proper sequencing, this can result in a two-fold performance improvement [Ganger94]. For more aggressive implementations (e.g., [Hagmann87, Chutani92, Ganger95]), this reduces complexity and the amount of book-keeping required.

Directory sizes

A potential down-side of embedded inodes is that the directory size can increase substantially. While making certain that an inode and its name remain in the same sector, three directory entries with embedded 128-byte inodes can be placed in each 512-byte sector. Fortunately, as shown in Figure 4, the number of directory entries is generally small. For example, 94% of all directories would require only one 8 KB block on our file servers. For many of these, embedded inodes actually fit into space that was allocated anyway (the minimum unit of allocation is a 1 KB fragment). Still, there are a few directories with many entries (e.g., several of over 1000 and one of 9000). For a large directory, embedded inodes could greatly increase the amount of data that must be read from disk in order to scan just the names in a directory (e.g., when adding yet another directory entry). We are not particularly concerned about this, since bandwidth is what disks are good at. However, if experience teaches us that large directories are a significant problem, one option is to use the external inode "file" for the inodes of such directories.

3.2 Grouping Small Files

Like embedded inodes, small file grouping is conceptually quite simple. C-FFS simply places several small files adjacently on the disk and read/writes the entire group as a unit. The three main issues that arise are identifying the disk locations that make up a group, allocating disk locations appropriately, and caching the additional group items before they have been requested or identified. We discuss each of these below.

Identifying the group boundaries

To better exploit available disk bandwidth, C-FFS moves groups of blocks to and from the disk at once rather than individually. To do so, it must be possible to determine, for any given block, whether it is part of a group and, if so, what other disk locations are in the same group. C-FFS does this by augmenting each inode with two fields identifying the start and length of the group.[4] Because grouping is targeted for small files, we don't see any reason to allow an inode to point to data in more than one group. In fact, C-FFS currently allows only the first block (the only block for 79% of observed files) of any file other than a directory to be part of a group. Identifying the boundaries of the group is a simple matter of looking in the inode, which must already be present in order to identify the disk location of the desired block.

Allocation for groups

Before describing what is different about the allocation routines used to group small files, we want to stress what is not different. Placement of data for large files remains unchanged and should exploit clustering technology [Peacock88, McVoy91]. Directories are also placed as in current systems, by finding a cylinder group with many free blocks. (The fast file system actually performs directory allocation based on the number of free inodes rather than the amount of free space. With embedded inodes, however, this becomes both difficult and of questionable value.)

The main change occurs in deciding where to place the first block (or fragment) of a file. The standard approach is to scan the freelist for a free block in the cylinder group that contains the inode, always starting from the beginning. As files come and go, the free regions towards the front of the cylinder group become fragmented. C-FFS, on the other hand, tries to incorporate the new file block (or fragment) into an existing group associated with the directory that names the file. This succeeds if there is free space within one of the groups (perhaps due to a previous file deletion or truncation) or if there is a free block outside of a group that can be incorporated without causing the group to exceed its maximum size (currently hard-coded to 64 KB in our prototype, which approximates the knees of the curves in Figure 2). To identify the groups associated with a particular directory, C-FFS exploits the fact that using embedded inodes gives us direct access to all of the relevant inodes (and thus their group information) by examining the directory's inode (which identifies the boundaries of the first group) and scanning the directory (ignoring any inodes for directories). If the new block extends an existing group, C-FFS again exploits embedded inodes to scan the directory and update the group information for other group members. If the new block cannot be added to an existing group, the conventional approach is used to allocate the block and a new group with one member is created.

One point worth stressing about explicit grouping is that it does not require all blocks within the boundaries of a group to belong to inodes in a particular directory. Although C-FFS tries to achieve this ideal, arbitrary files can allocate blocks that end up falling within the boundaries of an unrelated group. The decision to allow this greatly simplifies the management of group information (i.e., it is simply a hint that describes blocks that are hopefully related). Although this could result in suboptimal behavior, we believe that it is appropriate given the fact that reading/writing a few extra disk blocks has a small incremental cost and given the complexity of alternative approaches.

Multi-block directories are the exception to both the allocation routine described above and to the rule that only the first block of a file can be included in a group. Because scanning the contents of a directory is a common operation, C-FFS tries to ensure that all of its blocks are part of the first group associated with the directory (i.e., the one identified by the directory's inode). Fortunately, even with embedded inodes, most directories

[4] Although we have added fields to support grouping in the C-FFS prototype, it would seem reasonable to overload two of the indirect block pointers instead and simply disable grouping for large files.

(e.g., 94 percent for our file servers) will be less than one block in size. If a subsequent directory block must be allocated and can not be incorporated into the directory's first group, C-FFS selects one of the non-directory blocks in the first group, moves it (i.e., copies it to a new disk location and changes the pointer in the inode), and gives its location to the new directory block. Once again, this operation exploits embedded inodes to find a member of the first group from which to steal a block. Unfortunately, moving the group member's block requires two synchronous writes (one to write the new block and one to update the corresponding inode). A better metadata integrity maintenance mechanism (e.g., write-ahead logging or soft updates) could eliminate these synchronous writes. When no blocks are available for stealing, C-FFS falls back to allocating a block via the conventional approach (with a preference for a location immediately after the first group).

Cache functionality required

Grouping requires that the file block cache provide a few capabilities, most of which are not new. In particular, C-FFS requires the ability to cache and search for blocks whose higher level identities (i.e., file ID and offset) are not known. Therefore, our file cache is indexed by both disk address[5], like the original UNIX buffer cache, and higher-level identities, like the SunOS integrated caching and virtual memory system [Gingell87, Moran87]. C-FFS uses physical identities to insert newly-read blocks of a group into the cache without back-translating to discover their file/offset identities. Instead, C-FFS inserts these blocks into the cache based on physical disk address and an invalid file/offset identity. When a cache miss occurs for a file/offset that is a member of any group, C-FFS searches the cache a second time, by physical disk address (since it might have been brought in by a previous grouped access). If the block is present under this identity, C-FFS changes the file/offset identity to its proper value and returns the block without an additional disk read.

The ability to search the cache by physical disk address is also necessary when initiating a disk request for an entire group. When reading a group, C-FFS prunes the extent read based on which blocks are already in the cache (it would be disastrous to replace a dirty block with one read from the disk). When writing a group, C-FFS prunes the extent written to include only those blocks that are actually present in the cache and dirty.

Given that C-FFS requires the cache to support lookups based on disk address, C-FFS uses this mechanism (rather than clustering or grouping information) to identify additional dirty blocks to write out with any given block. With this scheme, the main benefit provided by grouping is to increase the frequency with which dirty blocks are physically adjacent on the disk. C-FFS uses scatter/gather support to deal with non-contiguity of blocks in the cache. Were scatter/gather support not present, C-FFS would rely on more complicated extent-based memory management to provide contiguous regions of physical memory for reading and writing of both small file groups and large file clusters.

[5] By physical disk address, we mean the address given to the device driver when initiating a disk request, as opposed to more detailed information about how the cylinder, surface and rotational offset at which the data are stored inside the disk.

Specification	Value
Capacity	1 GB
Cylinders	2700
Surfaces	9
Sectors/Track	84
Rotation Speed	5400 RPM
Head Switch	N/A
One-cyl Seek	1 ms
Average Seek	9/10.5 ms
Maximum Seek	22 ms

Table 2: Characteristics of the Seagate ST31200 drive.

4 Performance Evaluation

This section reports measurements of our C-FFS implementation, which show that it can dramatically improve performance. For small file activity (both reads and writes), we observe order of magnitude reductions in the number of disk requests and factor of 5–7 improvements in performance. We observe no negative effects for large file I/O. Preliminary measurements of application performance show improvements of 10–300 percent.

4.1 Experimental Apparatus

All experiments were performed on a PC with a 120 MHz Pentium processor and 32 MB of main memory. The disk on the system is a Seagate ST31200 (see Table 2). The disk driver, originally taken from NetBSD, supports scatter/gather I/O and uses a C-LOOK scheduling algorithm [Worthington94]. The disk prefetches sequential disk data into its on-board cache. During the experiments, there was no other activity on the system, and no virtual memory paging occurred. In all of our experiments, we forcefully write back all dirty blocks before considering the measurement complete. Therefore, our disk request counts include all blocks dirtied by a particular application or micro-benchmark.

The disk drive used in the experiments is several years old and can deliver only one-third to one-half of the bandwidth available from a state-of-the-art disk (as seen in Table 1). As a result, we believe that the performance improvements shown for embedded inodes and explicit grouping are actually conservative estimates. These techniques depend on high bandwidth to deliver high performance for small files, while the conventional approach depends on disk access times (which have improved much less).

The C-FFS implementation evaluated here is the default file system for the Intel x86 version of the exokernel operating system [Engler95]. It includes most of the functionality expected from an FFS-like file system. Its major limitations are that it currently does not support prefetching or fragments (the units of allocation are 4 KB blocks). Prefetching is more relevant for large files than the small files our techniques address and should be independent of both embedded inodes and explicit grouping. Fragments, on the other hand, are very relevant and we expect that they would further increase the value of grouping, because the allocator would explicitly attempts to allocate fragments within the group rather than taking the first available fragment or trying to optimize for growth by allocating new fragment blocks.

We compare our C-FFS implementation to itself with embedded inodes and explicit grouping disabled. Although this may raise questions about how solid our baseline is, we are comfortable that it is reasonable. Comparisons of our restricted C-FFS (without embedded inodes or explicit grouping) to OpenBSD's FFS on the same hardware indicate that our baseline is actually as fast or faster for most file system operations (e.g., twice as fast for file creation and writing, equivalent for deletion and reading from disk, and much faster when the static file cache size of OpenBSD limits performance). The performance differences are partially due to the system structure, which links the file system code directly into the application (thereby avoiding many system calls), and partially due to the file system implementation (e.g., aggressively clustering dirty file blocks based on physical disk addresses rather than logical relationships).

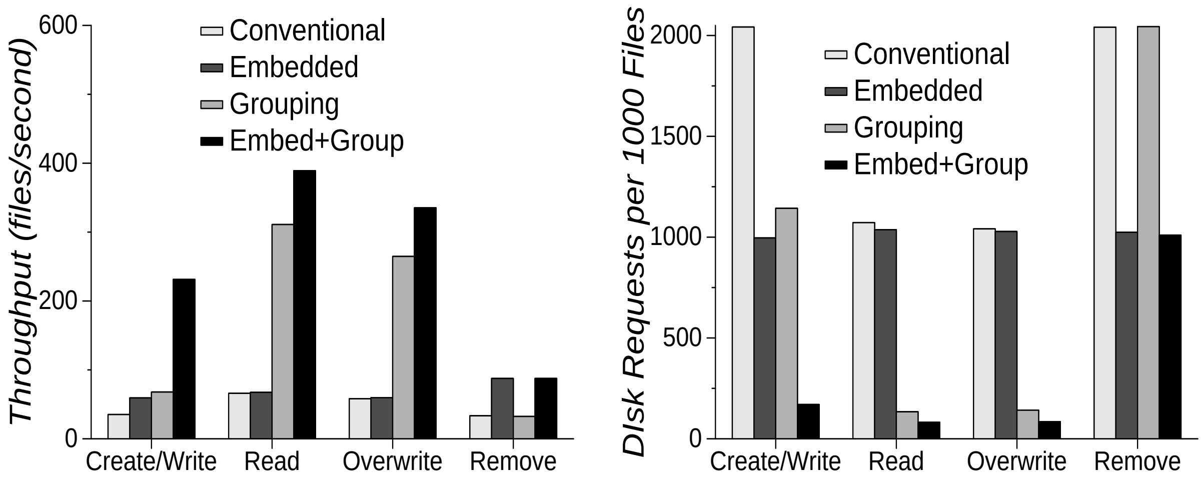
For our micro-benchmark experiments, we wanted to recreate the non-adjacency of on-disk placements observed on our file servers. Our simplistic approach was to modify the allocation routines. Rather than allocating a new file's first block at the first free location within the cylinder group, we start looking for free space at other locations within the cylinder group. For half of the files in a directory, we start looking at the last direct block of the previous entry. For the other half, we start looking at a random location within the cylinder group. The resulting data layouts are somewhat better than those observed on our file server, and therefore should favor the conventional layout scheme. We do not modify the inode allocation routines in any way, so inodes allocated in sequence for files in a given directory will tend to be packed into inode blocks much more densely than was observed on our server. Again, this favors the conventional organization. For the non-microbenchmark experiments, we do not do this.

4.2 Small File Performance

Figure 5 compares the throughput of 1 KB file operations supported by our prototype with embedded inodes, with explicit grouping, with neither and with both. The micro-benchmark, based on the small-file benchmark from [Rosenblum92], has four phases: create and write 10000 1KB files, read the same files in the same order, overwrite the same files in the same order, and then remove the files in the same order. To avoid the cost of name lookups in large directories, the files are spread among directories such that no single directory has more than 100 entries. In this comparison, both file systems are configured to use synchronous writes for metadata integrity maintenance (as is common among UNIX file systems).

For file creation, we observe a twentyfold reduction in the number of disk requests necessary when using both embedded inodes and explicit grouping. This results in a sevenfold increase in throughput. Half of the reduction in the number of disk requests comes from eliminating the synchronous writes required for integrity (by making the name and inode updates atomic) and half comes from writing the blocks for several new files with each disk request. It is interesting to note that half of the disk requests that remain are synchronous writes required because additional directory blocks are being forced into the directory's first group. This cost could be eliminated by a better integrity maintenance scheme (see below) or by not shuffling disk blocks in this way (which could involve a performance cost for later directory scanning operations). It is also interesting to observe that C-FFS with both embedded inodes and explicit grouping significantly outperforms C-FFS with either of these techniques alone (5 times fewer disk requests and 3–4 times the create/write throughput).

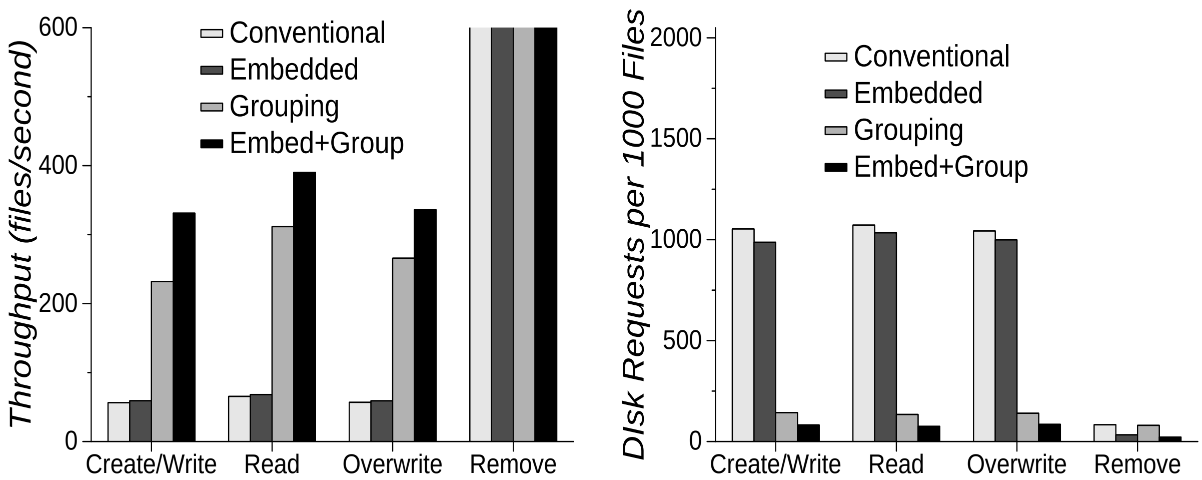
For file read and overwrite, we observe an order of magnitude reduction in the number of disk requests when using explicit grouping. This increases throughput by factors of 4.5–5.5. For these operations, embedded inodes alone provide marginal improvements. However, embedded inodes do provide measurable improvement when explicit grouping is in use, once again making the combination of the two the best option.



(a) Files per second

(b) Disk requests per 1000 files

Figure 5: Small file throughput when using synchronous writes for metadata integrity.



(a) Files/second

(b) Disk requests/1000 files

Figure 6: Small file throughput when using soft updates for metadata integrity. The “remove” throughput values are off the scale, ranging from 1500–2000 files per second.

For file deletion, we see a twofold decrease in the number of disk requests when using embedded inodes, since embedding the inodes eliminates one of the two sequencing requirements for file deletion. The second sequencing requirement, which relates to the relationship between the inode and the free map, remains. The result is a 250% increase in file deletion throughput, provided both by the reduction in the number of disk requests and improved locality (i.e., the same block gets overwritten repeatedly as the multiple inodes that it contains are re-initialized).

Because there exist techniques (e.g., soft updates [Ganger94]) that have been shown to effectively eliminate the performance cost of maintaining metadata integrity, we repeat the same experiments with this cost removed. Figure 6 shows these results. We have not yet actually implemented soft updates in C-FFS, but rather emulate it by using delayed writes for all metadata updates — [Ganger94] shows that this will accurately predict the performance impact of soft updates. The change in performance that we observe for the conventional file system is consistent with previous studies.

With the synchronous metadata writes removed, we still observe an order of magnitude reduction in disk requests when using explicit grouping for create/write, read and overwrite operations. The result is throughput increases of 4–7 times. Although the performance benefit of embedded inodes is lower for the create/write phase (because synchronous writes are not an issue), there is still a gain because embedded inodes significantly reduce the work involved with allocation for explicit grouping. As before, the combination of embedded inodes and explicit grouping provides the highest throughput for both the read and overwrite phases. With synchronous writes eliminated, file deletion throughput increases substantially. Although it has no interesting effect on performance (because resulting file deletion throughput is so high), embedding inodes halves the number of blocks actually dirtied when removing the files because there are no separate inode blocks.

4.3 File System Aging

To get a handle on the impact of file system fragmentation on the performance of C-FFS, we use an aging program similar to that described in [Herrin93]. The program simply creates and deletes a large number of files. The probability that the next operation performed is a file creation (rather than a deletion) is taken from a distribution centered around a desired file system utilization. After reaching the desired file system utilization for the first time, the aging program executes some number of additional file operations taken from the same distribution. The size of each file created is taken from the distribution measured on our file servers.

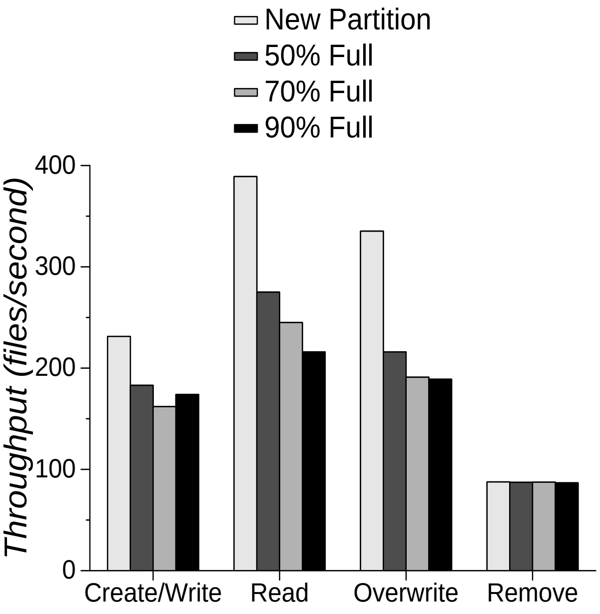


Figure 7: Small file throughput with embedded inodes and explicit grouping after aging the file system. Before running the small file micro-benchmark (without soft updates), the file system was aged by filling the file system to a desired capacity and then executing 100000 file create and delete operations. To increase the impact of the aging, we performed these experiments on a 128 MB partition. The four bars for each phase represent C-FFS performance for a fresh file system, for a 50%-full aged file system, for a 70%-full aged file system and for a 90%-full aged file system, respectively.

Figure 7 shows performance for the small file micro-benchmark after the file system has been aged. As expected, aging does have a significant negative impact on performance. At 70% capacity, the throughputs for the first three phases decrease by 30–40%. However, comparing these throughputs to those reported in the previous section, C-FFS still outperforms the conventional file system by a factor of 3–4. Further, our current C-FFS allocation algorithms do not reduce or compensate for fragmentation of free extents. We expect that the degradation due to file system aging can be significantly reduced by better allocation algorithms.

4.4 Large File Performance

Although C-FFS focuses on improving performance for small files, it is essential that it not reduce the performance that can be realized for large files. To verify that this is the case, we use a standard large file micro-

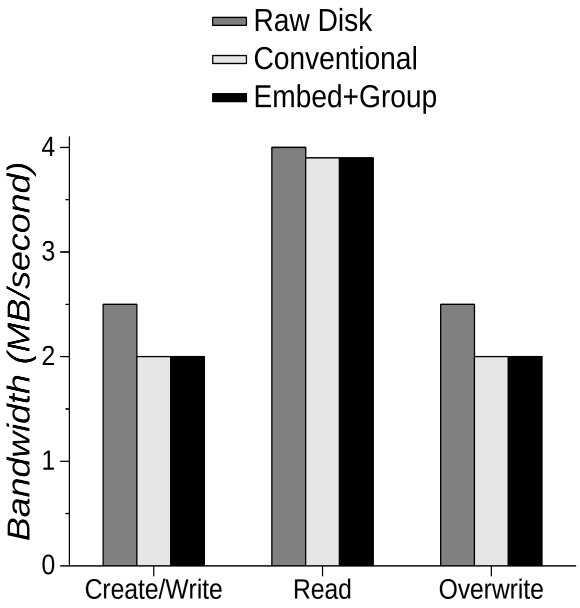


Figure 8: Large file bandwidth.

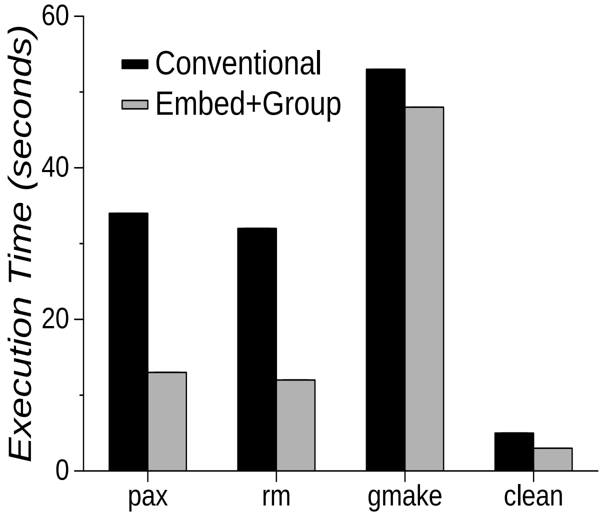


Figure 9: Application performance. "pax" uses the PAX utility to unpack a compressed archive file containing a substantial portion of the source tree for the default exokernel library operating system (about 1000 C files). "rm" uses the UNIX RM utility to recursively remove the directory tree created by "pax". "gmake" compiles a sub-directory containing 32 C files. "clean" removes the newly created object files from "gmake".

benchmark, which allocates and writes a large (32 MB) file sequentially, reads it sequentially and overwrites it sequentially. The results (shown in Figure 8) allow us to make two important points. First, using embedded inodes and explicit grouping has no significant effect on large file performance. Second, the C-FFS prototype, which supports clustering of large file data, delivers most of the disk's available bandwidth to applications for large files. (We believe that the somewhat disappointing file write bandwidth values are caused by software inefficiency; we have verified that it is not due to poor clustering.)

4.5 Applications

Figure 9 shows performance for four different applications which are intended to approximate some of the activities common to software development environments. As expected, we see significant improvements (e.g., a 50–66% reduction in execution time) for file-intensive applications like "pax" and "rm". For gmake, we see a much smaller improvement of only 10%. While such a small improvement could be viewed as a negative result, we were actually quite happy with it because of the extremely untuned nature of certain aspects of our system (which has just barely reached the point, at the time of this writing, where such applications can be run at all). In particular, process creation and shutdown (which are used extensively by the "gmake" application) are currently expensive. As a result, the time required for "gmake" on our exokernel system is currently 250% greater than the corresponding execution time on OpenBSD. The same absolute improvement given a baseline equal to that of OpenBSD would represent a 25% reduction in execution time for this compute-intensive task.

We believe that, for most applications, the name space provides useful information about relationships between files that can be exploited by explicit grouping. However, there are some applications (e.g., the web page caches of many HTTP browsers) that explicitly randomize file accesses across several directories in order to reduce the number of files per directory. For workloads where the name space is a poor indicator of access locality, we expect grouping to reduce read performance slightly, because it reads from the disk more data than are necessary.

5 Related Work

Previous researchers and file system implementors have been very successful at extracting large fractions of disks' maximum bandwidths for large files. One simple approach, which does have some drawbacks, is to increase the size of the basic file block [McKusick84]. Another is to cluster blocks (i.e., allocate them contiguously and read/write them as a unit when appropriate) [Peacock88, McVoy91]. Enumeration of a large file's disk blocks can also be significantly improved by using extents (instead of per-block pointers), B+ trees [Sweeney96] and/or sparse bitmaps [Herrin93]. We build on previous work by trying to exploit disk bandwidth for small files and metadata.

To increase the efficiency of the many small disk requests that characterize accesses to small files and metadata, file systems often try to localize logically related objects. For example, the Fast File System [McKusick84] breaks the file system's disk storage into cylinder groups and attempts to allocate most new objects in the same cylinder group as related objects (e.g., inodes in same cylinder group as containing directory and data blocks in same cylinder group as owning inode). Similarly, several researchers have investigated the value of moving the most popular (i.e., most heavily used) data to the centermost disk cylinders in order to reduce disk seek distances [Vongsathorn90, Ruemmler91, Staelin91]. As described in Section 2, simply locating related objects near each other offers some performance gains, but such locality affects only the seek time and is thus limited in scope. Co-locating related objects and reading/writing them as a unit offers qualitatively larger improvements in performance.

Immediate files are a form of co-location for small files. The idea, as proposed by [Mullender84], is to expand the inode to the size of a block and include the first part of the file in it. They found that over 60% of their files could be kept with the inode in a single block (and therefore both could be read or written with a single disk request). Another way to look at the same idea is simply that the inode is moved to the first block of the file. One potential down-side to this approach is that it replaces co-location of possibly related inodes (in an inode block) with co-location of an inode and its file data, which can reduce the performance of operations that examine file attributes but not file data. One simple application of the basic idea, however, is to put the data for very small files in the "normally" sized inode, perhaps replacing the space used by block pointers.

The log-structured file system's answer to the disk performance problem is to delay, remap and cluster all new data, only writing large chunks to the disk [Rosenblum92]. So long as neither cleaning nor read traffic represent significant portions of the workload, this will offer the highest performance. Unfortunately, while it may be feasible to limit cleaning activity to idle periods [Blackwell95], anecdotal evidence, measurements of real systems (e.g., [Baker91]), and simulation studies (e.g., [Dahlin94]) all suggest that main memory caches have not eliminated read traffic as hoped. Our work attempts to achieve performance improvements for both reads and writes of small files and metadata. While we are working within the context of a conventional update-in-place file system, we could easily see co-location being used in a log-structured file system to improve performance when reads become necessary.

One of the extra advantages of embedded inodes is the elimination of one sequencing constraint when creating and deleting files. There are several more direct and more comprehensive approaches to reducing the performance cost of maintaining metadata integrity, including write-ahead logging [Hagmann87, Chutani92, Journal92, Sweeney96], shadow-paging [Chamberlin81, Stonebraker87, Chao92, Seltzer93] and soft updates [Ganger95]. As shown in Section 4, our work complements such approaches.

Of course, there is a variety of other work that has improved file system performance via better caching, prefetching, write-back, indexing, scheduling and disk array mechanisms. We view our work as complementary to these.

6 Discussion

The C-FFS implementation described and evaluated in this paper is part of the experimental exokernel OS [Engler95]. We have found the exokernel to be an excellent platform for systems research of this kind. In particular, our first proof-of-concept prototype was extremely easy to build and test, because it did not have to deal with complex OS internals and it did not have to pay high overheads for being outside of the operating system.

However, the system-related design challenges for file systems in an exokernel OS (which focuses on distributing control of resources among applications) are different from those in a more conventional OS. An implementation of C-FFS for OpenBSD is underway, both to allow us to better understand how it interacts with an existing FFS and to allow us to more easily transfer it to existing systems. Early experience suggests that C-FFS changes some of the locking and buffer management assumptions made in OpenBSD, but there seem to be no fundamental roadblocks.

In this paper, we have compared C-FFS to an FFS-like file system, both with and without additional support for eliminating the cost of metadata integrity maintenance. Although we have not yet performed measurements, it is also interesting to compare C-FFS to other file systems (in particular, the log-structured file system). We believe that C-FFS can match the write performance of LFS, so long as the name space correctly indicates logical relationships between files. For read performance, the comparison is more interesting. C-FFS will perform best when read access patterns correspond to relationships in the name space. LFS will perform best when read access patterns exactly match write access patterns. Although experiments is needed, we believe that C-FFS is likely to outperform LFS in many cases, especially when multiple applications are active concurrently.

In this paper, we investigate co-locating files based on the name space; other approaches based on application-specific knowledge are worth investigating. For example, one application-specific approach is to group files that make up a single hypertext document [Kaashoek96]. We are investigating extensions to the file system interface to allow this information to be passed to the file system. The result will be a file system that groups files based on application hints when they are available and name space relationships when they are not.

Our experience with allocation for small file grouping is preliminary, and there are a variety of open questions. For example, although the current C-FFS implementation allows only one block from a file to belong to a group, we suspect that performance will be enhanced and fragmentation will be reduced by allowing more than one. Also, C-FFS currently allows a group to be extended to include a new block even if it must also include an unrelated block that had (by misfortune) been allocated at the current group's boundary. We believe, based on our underlying assumption that reading an extra block incurs a small incremental cost, that this choice is appropriate. However, measurements that demonstrate this and indicate a maximum size for such holes are needed. Finally, allocation algorithms that reduce and compensate for the fragmentation of free space caused by aging will improve performance significantly (by 40–60%, according to the measurements in Section 4.3).

7 Conclusions

C-FFS combines embedded inodes and explicit grouping of files with traditional FFS techniques to obtain high performance for both small and large file I/O (both reads and writes). Measurements of our C-FFS implementation show that the new techniques reduce the number of disk requests by an order of magnitude for standard small file activity benchmarks. For the system under test, this translates into performance improvements of a factor of 5–7. The new techniques have no negative impact on large file I/O; the FFS clustering still delivers maximal performance. Preliminary experiments with real applications show performance improvements of 10–300 percent.

Acknowledgement

We thank Kirk McKusick and Bruce Worthington for early feedback on these ideas. We thank Garth Gibson, John Wilkes and the anonymous reviewers for detailed feedback in the later stages. We also thank the other members of the Parallel and Distributed Operating Systems research group at MIT, especially Héctor Briceño, Dawson Engler, Anthony Joseph, Eddie Kohler, David Mazières, Costa Sapuntzakis, and Josh Tauber for their comments, critiques and suggestions. In particular, we thank Costa who, while implementating C-FFS for OpenBSD, has induced many detailed discussions of design issues related to embedded inodes, explicit grouping and file systems in general. We especially thank Héctor and Tom Pinckney for their efforts as the final deadline approached.

References

[Baker91] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, J. Ousterhout, "Measurements of a Distributed File System", *ACM Symposium on Operating Systems Principles*, 1991, pp. 198–212.

[Blackwell95] T. Blackwell, J. Harris, M. Seltzer, "Heuristic Cleaning Algorithms in Log-Structured File Systems", *USENIX Technical Conference*, January 1995, pp. 277-288.

[Chamberlin81] D. Chamberlin, M. Astrahan, et. al., "A History and Evaluation of System R", *Communications of the ACM*, Vol. 24, No. 10, 1981, pp. 632–646.

[Chao92] C. Chao, R. English, D. Jacobson, A. Stepanov, J. Wilkes, "Mime: A High-Performance Parallel Storage Device with Strong Recovery Guarantees", Hewlett-Packard Laboratories Report, HPL-CSP-92-9, November 1992.

[Chutani92] S. Chutani, O. Anderson, M. Kazar, B. Leverett, W. Mason, R. Sidebotham, "The Episode File System", *Winter USENIX Conference*, January 1992, pp. 43–60.

[Dahlin94] M. Dahlin, R. Wang, T. Anderson, D. Patterson, "Cooperative Caching: Using Remote Client Memory to Improve File System Performance", *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994, pp. 267–280.

[Engler95] D. Engler, M.F. Kaashoek, J. O'Toole Jr., "Exokernel: an operating system architecture for application-level resource management", *ACM Symposium on Operating Systems Principles*, Dec. 1995, pp. 251-266.

[Forin94] A. Forin, G. Malan, "An MS-DOS File System for UNIX", *Winter USENIX Conference*, January 1994, pp. 337-354.

[Ganger94] G. Ganger, Y. Patt, “Metadata Update Performance in File Systems”, *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994, pp. 49–60.

[Ganger95] G. Ganger, Y. Patt, “Soft Updates: A Solution to the Metadata Update Problem in File Systems”, Technical Report CSE-TR-254-95, University of Michigan, Ann Arbor, August 1995.

[Gingell87] R. Gingell, J. Moran, W. Shannon, “Virtual Memory Architecture in SunOS”, *Summer USENIX Conference*, June 1987, pp. 81–94.

[Hagmann87] R. Hagmann, “Reimplementing the Cedar File System Using Logging and Group Commit”, *ACM Symposium on Operating Systems Principles*, November 1987, pp. 155–162.

[Herrin93] E. Herrin, R. Finkel, “The Viva File System”, Technical Report #225-93, University of Kentucky, Lexington, 1993.

[HP91] Hewlett-Packard Company, “HP C2247 3.5-inch SCSI-2 Disk Drive – Technical Reference Manual”, Edition 1, December 1991.

[HP92] Hewlett-Packard Company, “HP C2244/45/46/47 3.5-inch SCSI-2 Disk Drive Technical Reference Manual”, Part Number 5960-8346, Edition 3, September 1992.

[HP96] Hewlett-Packard Company, http://www.dmo.hp.com/disks/oemdisk/c3653a.html, June 1996.

[Journal92] NCR Corporation, “Journaling File System Administrator Guide, Release 2.00”, NCR Document D1-2724-A, April 1992.

[Kaashoek96] M.F. Kaaskoek, D. Engler, G. Ganger, D. Wallach, “Server Operating Systems”, *ACM SIGOPS European Workshop*, September 1996, pp. 141-148.

[McKusick84] M. McKusick, W. Joy, S. Leffler, R. Fabry, “A Fast File System for UNIX”, *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 181–197.

[McKusick94] M. McKusick, T.J. Kowalski, “Fsck – The UNIX File System Check Program”, *4.4 BSD System Manager’s Manual*, O’Reilley & Associates, Inc., Sebastopol, CA, 1994, pp. 3:1–21.

[McVoy91] L. McVoy, S. Kleiman, “Extent-like Performance from a UNIX File System”, *Winter USENIX Conference*, January 1991, pp. 1–11.

[Moran87] J. Moran, “SunOS Virtual Memory Implementation”, *European UNIX Users Group (EUUG) Conference*, Spring 1988, pp. 285–300.

[Mullender84] S. Mullender, A. Tanenbaum, “Immediate Files”, *Software–Practice and Experience*, 14 (4), April 1984, pp. 365–368.

[Ousterhout85] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, J. Thompson, “A Trace-Driven Analysis of the UNIX 4.2 BSD File System”, *ACM Symposium on Operating System Principles*, 1985, pp. 15–24.

[Peacock88] J.K. Peacock, “The Counterpoint Fast File System”, *USENIX Winter Conference*, February 1988, pp. 243–249.

[Quantum96] Quantum Corporation, http://www.quantum.com/products/atlas2/, June 1996.

[Rosenblum92] M. Rosenblum, J. Ousterhout, “The Design and Implementation of a Log-Structured File System”, *ACM Transactions on Computer Systems*, 10 (1), February 1992, pp. 25-52.

[Rosenblum95] M. Rosenblum, E. Bugnion, S. Herrod, E. Witchel, A. Gupta, “The Impact of Architectural Trends on Operating System Performance”, *ACM Symposium on Operating Systems Principles*, Dec. 1995, pp. 285-298.

[Ruemmler91] C. Ruemmler, J. Wilkes, “Disk Shuffling”, Technical Report HPL-CSP-91-30, Hewlett-Packard Laboratories, October 3, 1991.

[Ruemmler93] C. Ruemmler, J. Wilkes, “UNIX Disk Access Patterns”, *Winter USENIX Conference*, January 1993, pp. 405–420.

[Seagate96] Seagate Technology, Inc., http://www.seagate.com/stor/stortop.shtml, June 1996.

[Seltzer93] M. Seltzer, K. Bostic, M. McKusick, C. Staelin, “An Implementation of a Log-Structured File System for UNIX”, *Winter USENIX Conference*, January 1993, pp. 201–220.

[Seltzer95] M. Seltzer, K. Smith, M. Balakrishnan, J. Chang, S. McMains, V. Padmanabhan, “File System Logging Verses Clustering: A Performance Comparison”, *USENIX Technical Conference*, January 1995, pp. 249-264.

[Smith96] K. Smith, M. Seltzer, “A Comparison of FFS Disk Allocation Policies”, *USENIX Technical Conference*, January 1996, pp. 15-25.

[Staelin91] “Smart Filesystems”, *Winter USENIX Conference*, 1991, pp. 45–51.

[Stonebraker81] M. Stonebraker, “Operating System Support for Database Management”, *Communications of the ACM*, 24 (7), 1981, pp. 412–418.

[Stonebraker87] M. Stonebraker, “The Design of the POSTGRES Storage System”, *Very Large Data Base Conference*, September 1987, pp. 289–300.

[Sweeney96] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, G. Peck, “Scalability in the XFS File System”, *USENIX Technical Conference*, 1996, pp. 1–14.

[Vongsathorn90] P. Vongsathorn, S. Carson, “A System for Adaptive Disk Rearrangement”, *Software – Practice and Experience*, Vol. 20, No. 3, March 1990, pp. 225–242.

[Worthington94] B. Worthington, G. Ganger, Y. Patt, “Scheduling Algorithms for Modern Disk Drives”, *ACM SIGMETRICS Conference*, May 1994, pp. 241–251.

[Worthington95] B. Worthington, G. Ganger, Y. Patt, J. Wilkes, “On-Line Extraction of SCSI Disk Drive Parameters”, *ACM SIGMETRICS Conference*, May 1995, pp. 146–156.